*To read the article online, visit http://www.4GuysFromRolla.com/articles/101106-1.aspx*

# Examining ASP.NET's Membership, Roles, and Profile - Part 6

By Scott Mitchell

## Introduction

The Membership API in the .NET Framework provides the concept of a user account and associates with it core properties: username, password, email, security question and answer, whether or not the account has been approved, whether or not the user is locked out of the system, and so on. However, depending on the application's needs, chances are your application needs to store additional, user-specific fields. For example, an online messageboard site might want to also allow users to specify a signature, their homepage URL, and their IM address.

There are two ways to associate additional information with user accounts when using the Membership model. The first - which affords the greatest flexibility, but requires the most upfront effort - is to create a custom data store for this information. If you are using the SqlMembershipProvider, this would mean creating an additional database table that had as a primary key the `UserId` value from the `aspnet_Users` table and columns for each of the additional user properties. In the online messageboard example, the table might be called `forums_UserProfile` and have columns like `UserId` (a primary key *and* a foreign key back to `aspnet_Users.UserId`), `HomepageUrl`, `Signature`, and `IMAddress`.

Rather than using custom data stores, the ASP.NET *Profile system* can be used to store user-specific information. The Profile system allows the page developer to define the properties she wants to associate with each user. Once defined, the developer can programmatically read from and assign values to these properties. The Profile system accesses or writes the property values to a backing store as needed. Like Membership and Roles, the Profile system is based on the provider model, and the particular Profile provider is responsible for serializing and deserializing the property values to some data store. The .NET Framework ships with a `SqlProfileProvider` class by default, which uses a SQL Server database table (`aspnet_Profile`) as its backing store.

In this article we will examine the Profile system - how to define the user-specific properties and interact with them programmatically from an ASP.NET page - as well as look at using the `SqlProfileProvider` that ships with .NET 2.0. In a future article we'll look at how to create and use a custom profile provider. Read on to learn more!

## An Overview of the Profile System

The Membership system in ASP.NET was designed to create a standardized API for working with user accounts, a task faced by many web applications (refer back to Part 1 of this article series for a more in-depth look at Membership). While the Membership system encompasses core user-related properties - username, password, email address, and so on - oftentimes additional information needs to be captured for each user. Unfortunately, this additional information can differ wildly from application to application.

Rather than add additional user attributes to the Membership system, Microsoft instead created the *Profile system* to handle additional user properties. The Profile system allows the additional, user-specific properties to be defined in the `Web.config` file and is responsible for persisting these values to some data store. How the user-specific properties are serialized and deserialized to a backing store is the responsibility of the configured *Profile provider* (see A Look at ASP.NET 2.0's Provider Model for more information on the provider model and its uses). The default Profile provider - `SqlProfileProvider` -

serializes the property values to the `aspnet_Profile` table in a SQL Server database.

When working with the Profile system, keep in mind that it's sole purpose is to serve as a means for defining a set of user-specific properties and, through a particular provider, serialize those property values to some backing store.

**Defining the User-Specific Properties**

With the Profile system, the user-specific properties must be spelled out in the `Web.config`. For each property, you can specify the name, data type, and how the data should be serialized. There are four serialization options:

- `ProviderSpecific` (the default) - the Profile provider is tasked with determining how to serialize the property value
- `String` - the property value is converted to a string representation when persisted by the Profile provider;
- `Xml` - the property value is converted to an XML representation when persisted by the Profile provider
- `Binary` - the property value is converted to a binary representation when persisted by the Profile provider

What serialization method you choose typically depends upon the type of the variable. You can instruct the provider to serialize the data as a String if there is a type converter for the property's data type that allows transformation to a String (such type converters exist for all of the primitive types - Strings, Booleans, Integers, and so on). To serialize as XML, the type must be XML serializable, and to serialize as Binary, the type must be marked serializable. (For more on XML serialization, see XML Serialization in ASP.NET; for info on the binary object serialization, see C# Object Serialization.)

The user-specific properties defined in the Profile system can be simple scalar properties, scalar properties grouped into complex types, or based on pre-existing complex types (such as a class). The names, types, and serialization instructions for each Profile property is spelled out in the `<profile>` element's `<properties>` section. For example, imagine that we were creating an online messageboard and wanted to allow each user to specify a homepage URL that would appear alongside their posts. To capture this information, we could add the following profile property:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <system.web>
      ...

      <profile defaultProvider="CustomProfileProvider" enabled="true">
        <providers>
          ...
        </providers>

        <!-- Define the properties for Profile... -->
        <properties>
          <add name="HomepageUrl" type="String" serializeAs="String" />

          ...
        </properties>
      </profile>
    </system.web>
</configuration>
```

The `<add>` element adds a Profile property named `HomepageUrl` of type String and serialized as a String.

There are additional attributes that can be included in the `<add>` element, such as `defaultValue` and `readOnly`, among others. See the technical documentation on the `<add>` element for more information.

Scalar profile properties can be grouped using the `<group>` element. For example, in addition to the `HomepageUrl` property we might also want to capture biographical information for our users, including their physical location, birthdate, and programming language of choice. This information can appear as three scalar values, or can be grouped. The following markup shows how to group these three properties into a group named `Bio`.

```
<profile defaultProvider="CustomProfileProvider" enabled="true">
  <providers>
    ...
  </providers>

  <!-- Define the properties for Profile... -->
  <properties>
    <add name="HomepageUrl" type="String" serializeAs="String" />

    <group name="Bio">
      <add name="BirthDate" type="DateTime" serializeAs="Xml" />
      <add name="Location" type="String" />
      <add name="ProgrammingLanguageOfChoice" type="ProgrammingLanguages"  />
    </group>

    ...
  </properties>
</profile>
```

In this grouping, the `BirthDate` property (of type DateTime) was chosen to be serialized as XML. The `Location` and `ProgrammingLanguageOfChoice` properties don't have a `serializeAs` attribute specified, meaning that the Profile provider being used will ascertain how to serialize these properties. Note the `type` for the `ProgrammingLanguageOfChoice` property. This is a type I created in the `App_Code` folder. Specifically, it's an enumeration that's defined with the following code:

```
Public Enum ProgrammingLanguages As Integer
    NoneSelected = 0
    VB = 1
    CSharp = 2
    JSharp = 3
End Enum
```

Not only can Profile properties have enumerations as their type, but also can have custom classes as their type. In the project that can be downloaded at the end of this article you'll find a very simple `Address` class (also in the `App_Code` folder), that has properties representing an address:

```
<Serializable()> _
Public Class Address
    Public Address1 As String
    Public Address2 As String
    Public City As String
    Public State As String
    Public Zip As Integer
End Class
```

With this class defined, I can add Profile properties of type `Address` like so:

```
<profile defaultProvider="CustomProfileProvider" enabled="true">
```

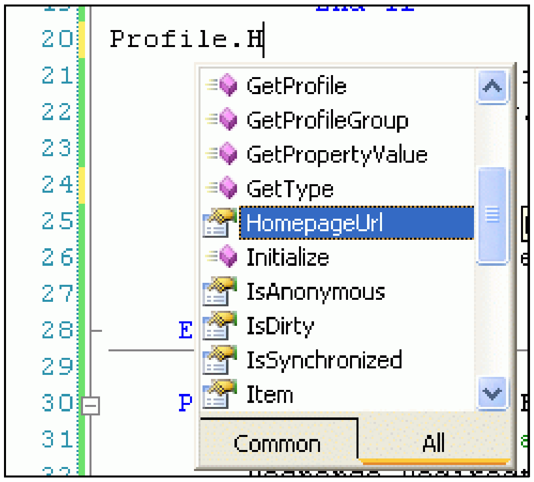
```
    <providers>
      ...
    </providers>

    <!-- Define the properties for Profile... -->
    <properties>
      <add name="HomepageUrl" type="String" serializeAs="String" />

      <group name="Bio">
        <add name="BirthDate" type="DateTime" serializeAs="Xml" />
        <add name="Location" type="String" />
        <add name="ProgrammingLanguageOfChoice" type="ProgrammingLanguages"  />
      </group>

      <add name="BillingAddress" type="Address" serializeAs="Xml" />
      <add name="ShippingAddress" type="Address" serializeAs="String" />
    </properties>
  </profile>
```

Both the `BillingAddress` and `ShippingAddress` properties are of type `Address`. `BillingAddress` is serialized as XML, while `ShippingAddress` is serialized using Binary serialization. (In order to support binary serialization, the property's type must implement `ISerializable` or have the `Serializable` attribute applied.)

## Working with the Profile Properties in Code

After defining your Profile properties, the ASP.NET engine automatically creates a `ProfileCommon` class that has properties that map to the properties defined in `Web.config` (which allows for IntelliSense and compile-time type checking). This class is dynamically recreated whenever the properties defined in `Web.config` are modified and automatically works with the currently logged on user's profile. (You can also turn on profile support for anonymous users, and later migrate their profile data once they've logged on. But this topic is beyond the scope of this article, and will have to wait for a future installment of this series!)

The custom `ProfileCommon` class is accessible in the code portion of an ASP.NET page through the `HttpContext` object's `Profile` property. For example, to read the currently logged on user's `HomepageUrl` property from an ASP.NET page, simply use `Profile.HomepageUrl`. In fact, as soon as you type in `Profile` and hit period, IntelliSense brings up the various properties. Cool, eh? And the grouping functionality works as you'd expect. Type in `Profile` and hit period, and one of the properties is `Bio`. Enter `Bio` and hit period, and the three scalar properties appear in IntelliSense (`BirthDate`, `Location`, and `ProgrammingLanguageOfChoice`).



```
Profile.H
```

The download available at the end of this article has three Web pages worth examining. The first is `/UsersOnly/Default.aspx` and shows the currently logged on user's Profile information. The second page, `/UsersOnly/UpdateProfile.aspx`, allows the currently logged on user to update his Profile data. The third page demonstrates the `Profile` property's `GetProfile("`*`username`*`")` method, which can be used to get the Profile data for a user other than the currently logged-on user. The page `/UserList.aspx` lists all of the users in the system, including their membership information (their username, email, and so on) and `HomepageUrl` Profile property value.

## Specifying a Profile Provider

If you don't explicitly specify a Profile provider, the default provider - `SqlProfileProvider` - is used. This default provider stores each user's property values in the `aspnet_Profile` table. The `aspnet_Profile` stores each users profile settings in a single row in the table, using a rather inefficient means for storing the property names, data types, and values. The table's schema follows:

| **aspnet_Profile** | |
|---|---|
| UserId | uniqueidentifier; PK; FK to aspnet_Users |
| PropertyNames | ntext |
| PropertyValuesString | ntext |
| PropertyValuesBinary | image |
| LastUpdatedDate | datetime |

For a given record, the `PropertyNames` column contains each of the Profile property names along with how they are serialized and where in the `PropertyValuesString` or `PropertyValuesBinary` columns their values can be found. The values for each property are squished into the `PropertyValuesString` or `PropertyValuesBinary` columns, with String- and XML-serialized properties tucked in `PropertyValuesString` and Binary-serialized properties jammed in `PropertyValuesBinary`.

With our Profile properties, the `PropertyNames` column for a user will have a value like:

```
Bio.ProgrammingLanguageOfChoice:S:0:92:HomepageUrl:S:92:30:ShippingAddress:B:0:212:...
```

As you can see, each property name has the format ***PropertyName*:B|S:*StartIndex*:*Length***, with the **B** or **S** indicating whether the property's value can be found in `PropertyValuesString` (**S**) or `PropertyValuesBinary` (**B**). For example, the `Bio.ProgrammingLanguageOfChoice` property value is stored in `PropertyValuesString` (**S**) starting at position 0 with a length of 92. The `HomepageUrl` property is also serialized as a String and its value can be found at index 92 with a length of 30.

Looking at the `PropertyValuesString` column (where the String- and XML-serialized property values are stored), we find:

```
<?xml version="1.0" encoding="utf-16"?>
<ProgrammingLanguages>CSharp</ProgrammingLanguages>http://www.datawebcontrols.com...
```

The value for the `ProgrammingLanguageOfChoice` has been serialized as XML, starts at index 0, and is 92 characters long. Likewise, the `HomepageUrl` property starts at index 92 and has a length of 30 characters. (The `ShippingAddress` property uses Binary-serialization, and therefore is found in the `PropertyValuesBinary` table. As you can see from inspecting the `PropertyNames` column, the `ShippingAddress` value begins at index 0 and has a length of 212 bytes.)

All of the work of serializing Profile properties to the database is handled automatically by the

`SqlProfileProvider`. The table and stored procedures used by the `SqlProfileProvider` are automatically created when implementing the `SqlMembershipProvider` (see Part 1).

**Performance and Querying Limitations of the `SqlProfileProvider`**

Since the `SqlProfileProvider` squishes all of the Profile property values into one or two columns in a single row, each time *any* Profile property is read or written to, *all* of the Profile values must be serialized or deserialized. Additionally, the format of the data makes it next to impossible to query or report on the Profile information. Imagine that we wanted to determine how many users in our system used C#. Since that information is jammed into the `PropertyValuesString` column, determining this information would require expensive string processing or some routine that first "unpacks" the data into a normalized structure.

An alternative to the built-in `SqlProfileProvider` is the free Table Profile Provider created by Microsoft employee Hao Kung. This provider stores each Profile property as a separate database table column. Alternatively, you could create your own custom Profile provider that operates against existing database tables or uses some other data store (such as XML files, text files, Web services, and so on).

## Conclusion

ASP.NET's Membership system makes creating and managing user accounts a piece of cake. Unfortunately, the Membership system only defined the most basic of user-specific properties - username, password, email address, and so on. If you need to capture additional information about your users, you must either use a custom solution (create your own data store, write code to read and write data to this store, and so on) or use the Profile system. The Profile system allows page developers to define a set of user-specific properties in `Web.config`. This information is automatically translated into a class (`ProfileCommon`), which is accessible through the `HttpContext` class's `Profile` property.

ASP.NET ships with a default Profile provider, `SqlProfileProvider`, which stores the Profile property values in the `aspnet_Profile` table in a SQL Server database. This provide automatically handles all of the serialization and deserialization work, you just need to programmatically read from and write to the user's Profile data through the `Profile` property.

Happy Programming!

- By Scott Mitchell

## Attachments

- Download the code used in this article

| Article Information | |
|---|---|
| Article Title: | ASP.NET.Examining ASP.NET's Membership, Roles, and Profile - Part 6 |
| Article Author: | Scott Mitchell |
| Published Date: | October 11, 2006 |
| Article URL: | http://www.4GuysFromRolla.com/articles/101106-1.aspx |